Information:

21046698792-10889878942

Kaan Erkuş-Giray Berk Kuşhan

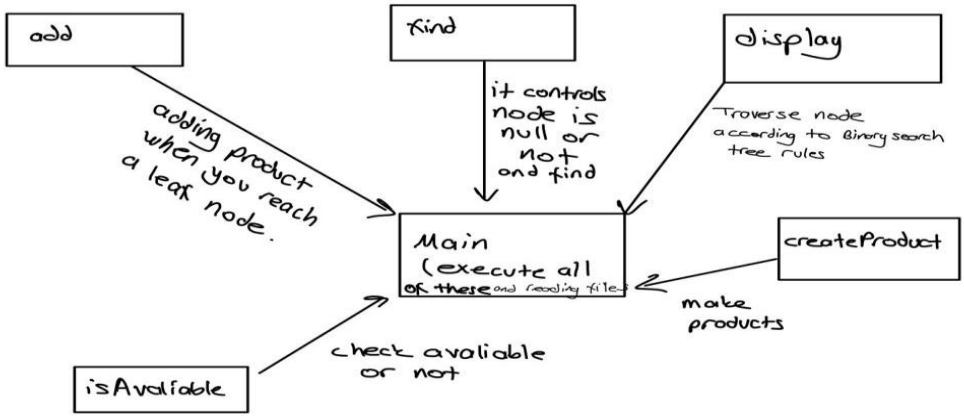
Sec-04

Homework3

Q1:

Problem Statement and Code Design:

In this assignment we created a project that Inventory Tracking System by implementing Binary Search Tree algorithm. The aim of the project is to create product and store it in our Binary Search Tree. Also, our code includes several sub-parts to make it modular. These sub modules are basically defined using a structure chart below.



Implementation Functionality:

To explain functionality of this program, we need to explain how binary search tree algorithm works. Binary search trees (BSTs) are a type of data structure that can be used to store and retrieve data efficiently. They are called binary search trees because each node in the tree has at most two children, and the children are organized in a specific way so that it is possible to search for a particular element in the tree quickly.

This project is occurred with 4 classes in one public class. These are Product, Node, SearchTree, Main and many methods.

add: If we reach leaf Node, we add the product. After that, we compare product's İd. If product's id less than current Node's product, we add to left. Otherwise, we add to right.

find: With this method, we control node is whether is null or not. And after that find.

display: We traverse the left and right subtree, and we visit the root node.

Creating Product: Thanks to this method we make products.

Checking is Available or not: This method checks available or not.

```java
class SearchTree{

    // Root of the tree
    Node root;


    public SearchTree(){

        // init root as null
        // Tree is empty
        this.root = null;
    }


    public Node add(Node node,Product product){
        // If we reach a leaf node, add the product
        if (node == null) {
            node = new Node(product);

            return node;
        }

        // If the product's ID is less than the current node's product, add left
        if (product.id < node.product.id) {

            node.leftNode = add(node.leftNode, product);
        }
```

```java
public class Main{

    SearchTree productDatabase;

    public Main()
    {
        productDatabase = new SearchTree();
    }


    public void createProduct(Product product){
        this.productDatabase.add(product);
    }

    public Node isAvaliable(int id){
        return this.productDatabase.find(this.productDatabase.root, id);

    }


    public static void main(String[] args) throws FileNotFoundException {
```

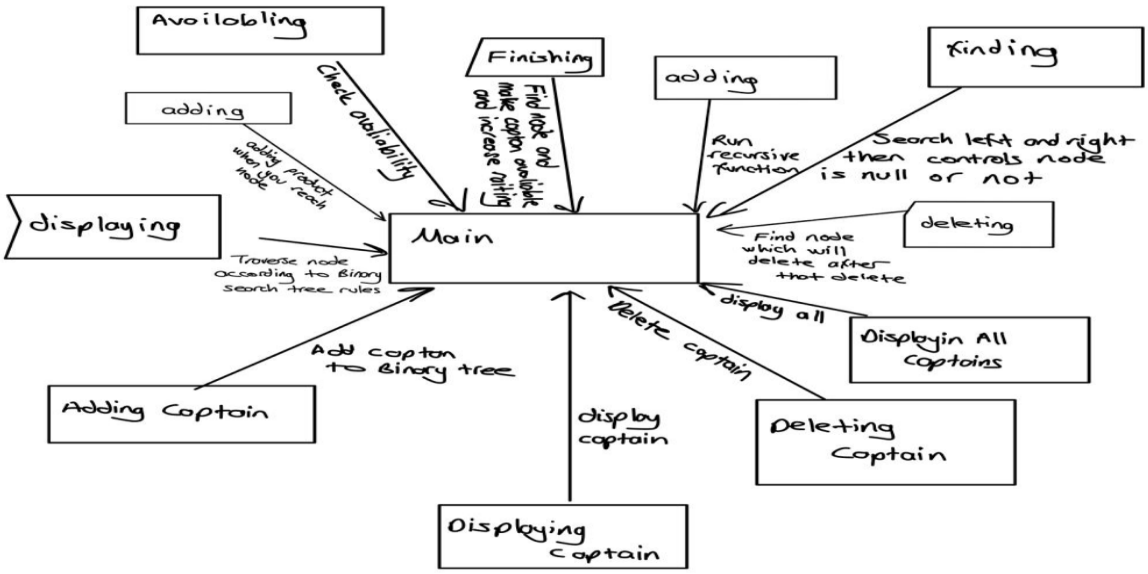
Testing:

There are 3 distinctive test cases that are given in the VPL. Test1 is a simple test case because products quantity is small. In test 2, vpl controls there are 9 products and two out of stock. So, this is a little bit harder than first test. In third test, there are 10 products, so it was the hardest one between the other test cases.

Q2:

Problem Statement and Code Design:

In this question we created a project that chauffeur-driven rental company (CDRC) which is asking management system code by implementing Binary Search Tree algorithm same with question 1. The aim of the project is to create management system code for their captain's salary increase. Also, our code includes several sub-parts to make it modular. These sub modules are basically defined using a structure chart below.



Implementation Functionality:

To explain functionality of this program, we need to explain how binary search tree algorithm works. Binary search trees (BSTs) are a type of data structure that can be used to store and retrieve data efficiently.

This question is occurred by 4 classes in one public class. These are Captain, Node, SearchTree, Main and many methods.

add: If we reach leaf Node, we add the product. After that, we compare product's İd. If product's id less than current Node's product, we add to left. Otherwise, we add to right.

Add: It is running recursive function

find: With this method, we control node is left and right parts whether is null or not. And after that find.

Delete: Find the node which will delete. After this, situation it is carrying out deleting part.

display: We traverse the left and right subtree, and we visit the root node.

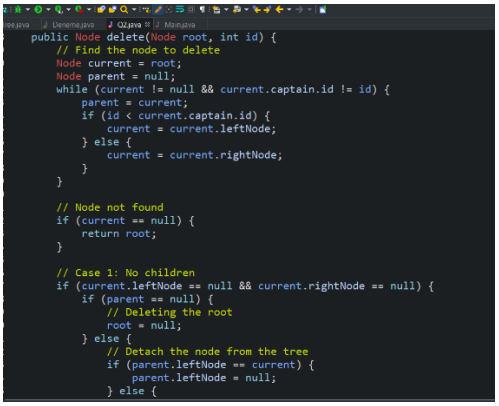
Checking is Available or not: This method checks available or not.

Adding Captain: Add captain to binary tree by concerning binary search tree rules.

Deleting Captain: Delete captain to binary tree.

Display Captain: Display captain to binary tree.

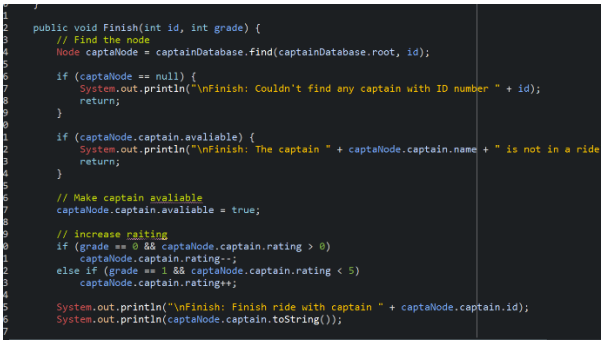
Finish: Find the node. Then make captain available and increase rating.



```
public Node delete(Node root, int id) {
    // Find the node to delete
    Node current = root;
    Node parent = null;
    while (current != null && current.captain.id != id) {
        parent = current;
        if (id < current.captain.id) {
            current = current.leftNode;
        } else {
            current = current.rightNode;
        }
    }

    // Node not found
    if (current == null) {
        return root;
    }

    // Case 1: No children
    if (current.leftNode == null && current.rightNode == null) {
        if (parent == null) {
            // Deleting the root
            root = null;
        } else {
            // Detach the node from the tree
            if (parent.leftNode == current) {
                parent.leftNode = null;
            } else {
```



```
public void Finish(int id, int grade) {
    // Find the node
    Node captaNode = captainDatabase.find(captainDatabase.root, id);

    if (captaNode == null) {
        System.out.println("\nFinish: Couldn't find any captain with ID number " + id);
        return;
    }

    if (captaNode.captain.avaliable) {
        System.out.println("\nFinish: The captain " + captaNode.captain.name + " is not in a ride
        return;
    }

    // Make captain avaliable
    captaNode.captain.avaliable = true;

    // increase raiting
    if (grade == 0 && captaNode.captain.rating > 0)
        captaNode.captain.rating--;
    else if (grade == 1 && captaNode.captain.rating < 5)
        captaNode.captain.rating++;

    System.out.println("\nFinish: Finish ride with captain " + captaNode.captain.id);
    System.out.println(captaNode.captain.toString());
```

Testing:

There are two test cases for second question. In the first test case, add captain, check avilability and if we could not find any captain write to console. So this is hard part of question 2 since we have a lot of captain. In second test case, we did same operations like first test case. But this is easier than first because we have less captain than first.

Final Assesments:

For final Assesment both parts generally have same troubles because we used binary search trees in both questions.

- We spent time finding the best algorithm as we thought we should use what kind of search tree for the particular problem We're working on.
- During my research, we learned about various path-finding algorithms such as binary search tree and the other tree models, but ultimately determined that binary search tree was the most effective option for our task.
- The most difficult part of the process for us was learning how can add members to binary tree. We search this problem and we found solution.